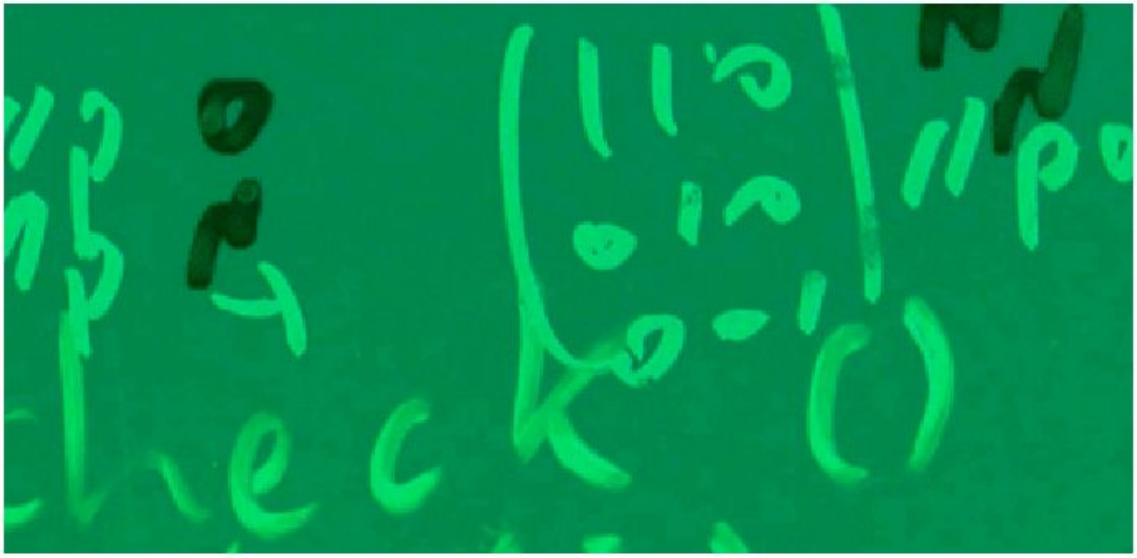
Algorithms and Data Structures to Accelerate Network Analysis

Reservoir Labs

Jordi Ros-Giralt, Alan Commike, Peter Cullen, Richard Lethin
{giralt, commike, cullen, lethin}@reservoir.com

4th International Workshop on Innovating the Network for Data Intensive Science

November 12, 2017



Reservoir Labs

632 Broadway
Suite 803
New York, NY 10012

Roadmap

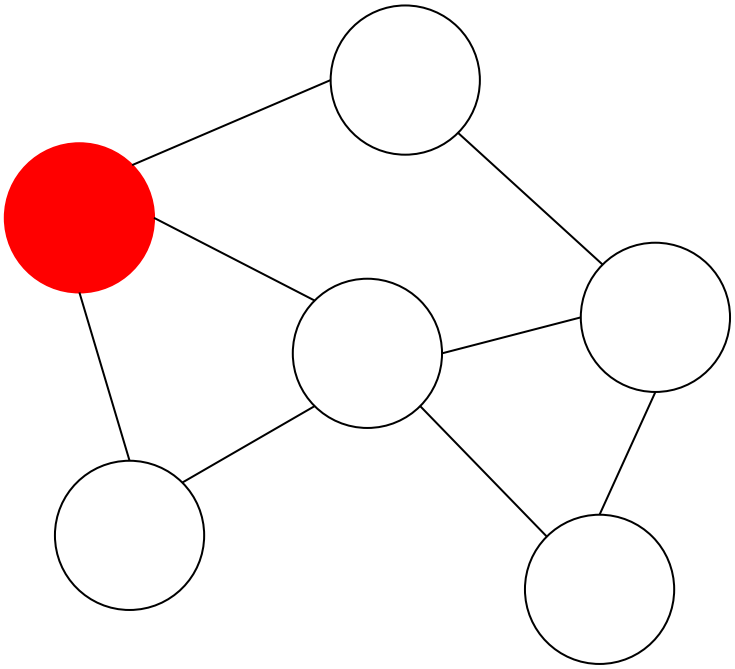
- Problem definition
- Optimizations
  - Long queue emulation
  - Lockless bimodal queues
  - Tail early dropping
  - LFN tables
  - Multiresolution priority queues
- Benchmarks

Problem Definition

- System wide optimization of network components like routers, firewalls, or network analyzers is complex.
- Hundreds of different SW algorithms and data structures interrelated in subtle ways.
- Two inter-related problems:
  - Shifting micro-bottlenecks
  - Nonlinear performance collapse

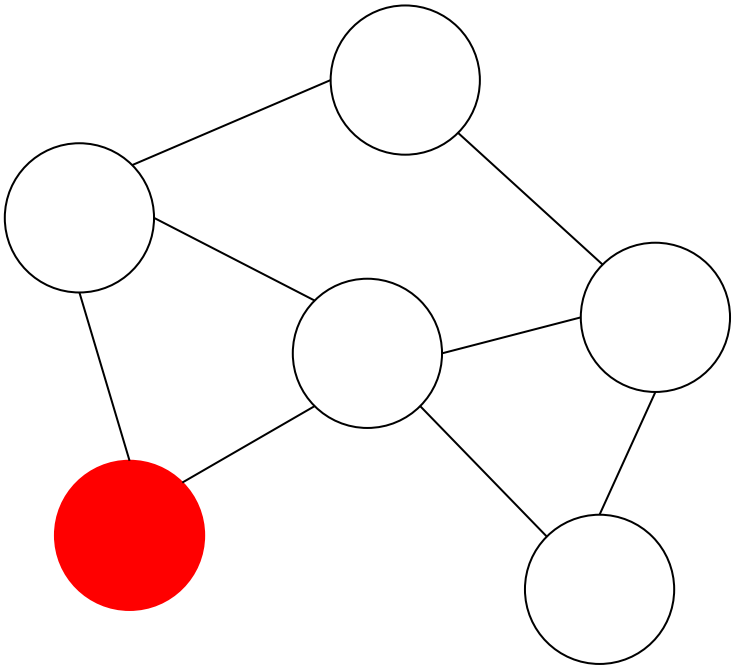
Shifting Micro-Bottlenecks

It’s difficult...



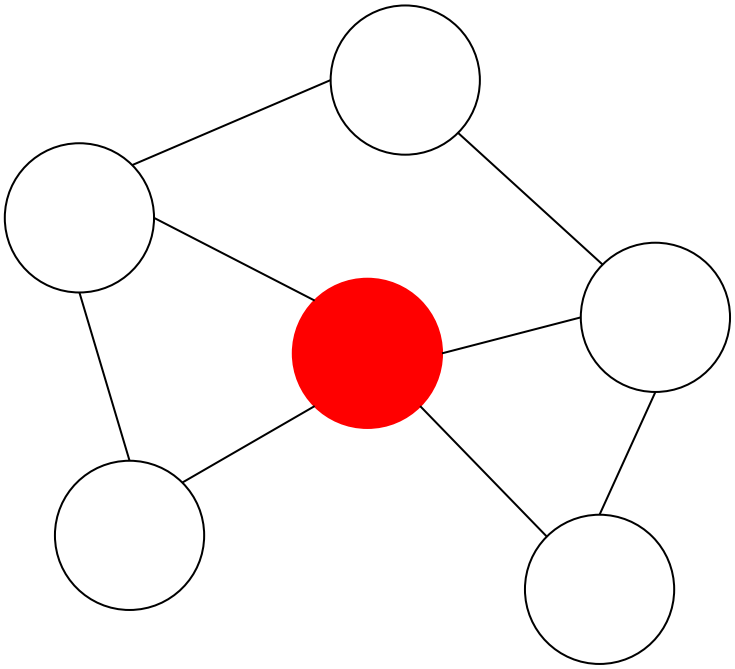
Shifting Micro-Bottlenecks

...to optimize...



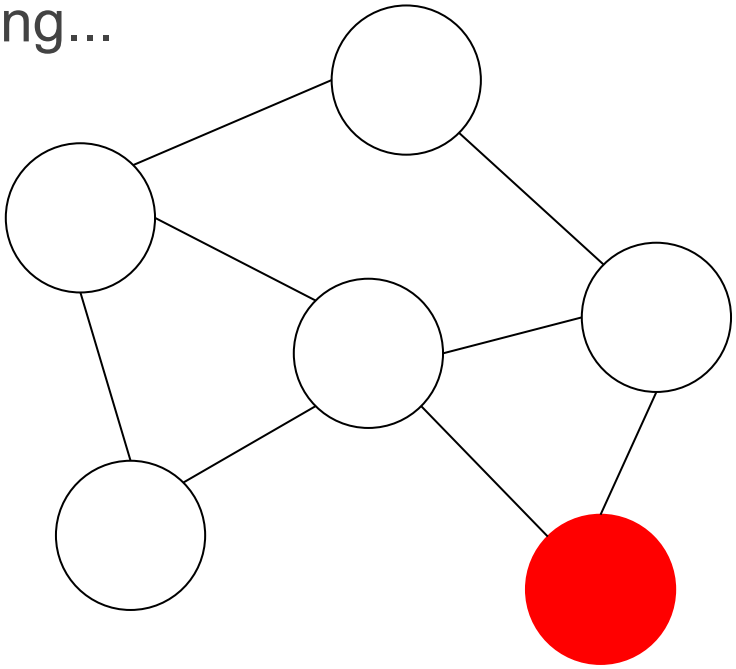
Shifting Micro-Bottlenecks

...bottlenecks...



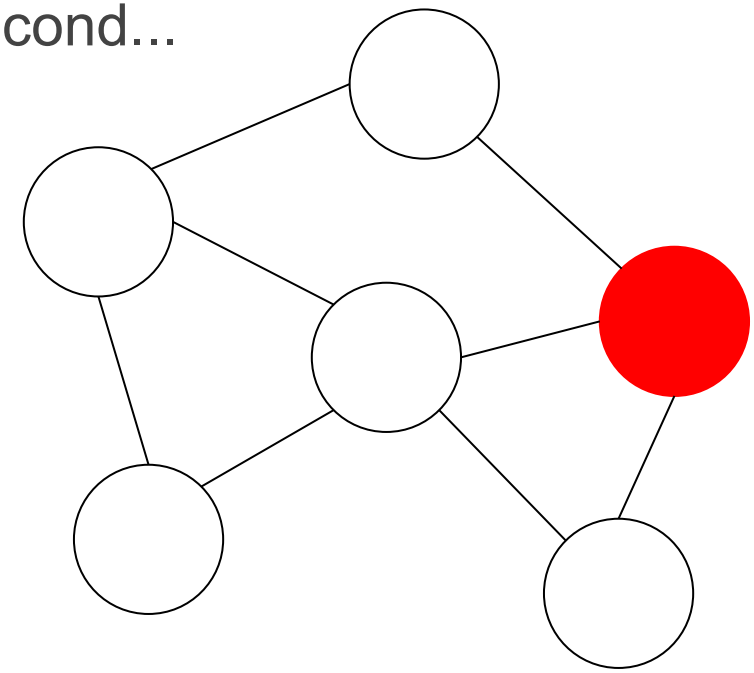
Shifting Micro-Bottlenecks

...that keep moving...



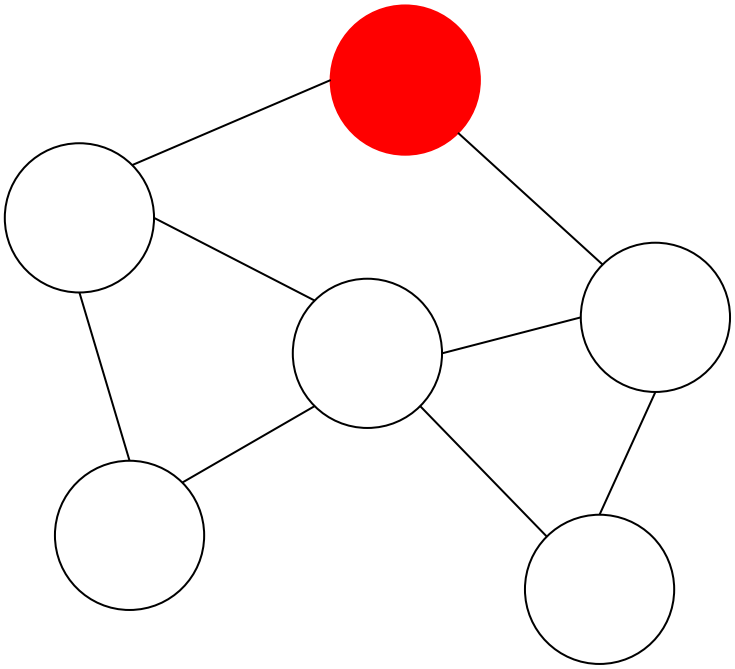
Shifting Micro-Bottlenecks

...every microsecond...

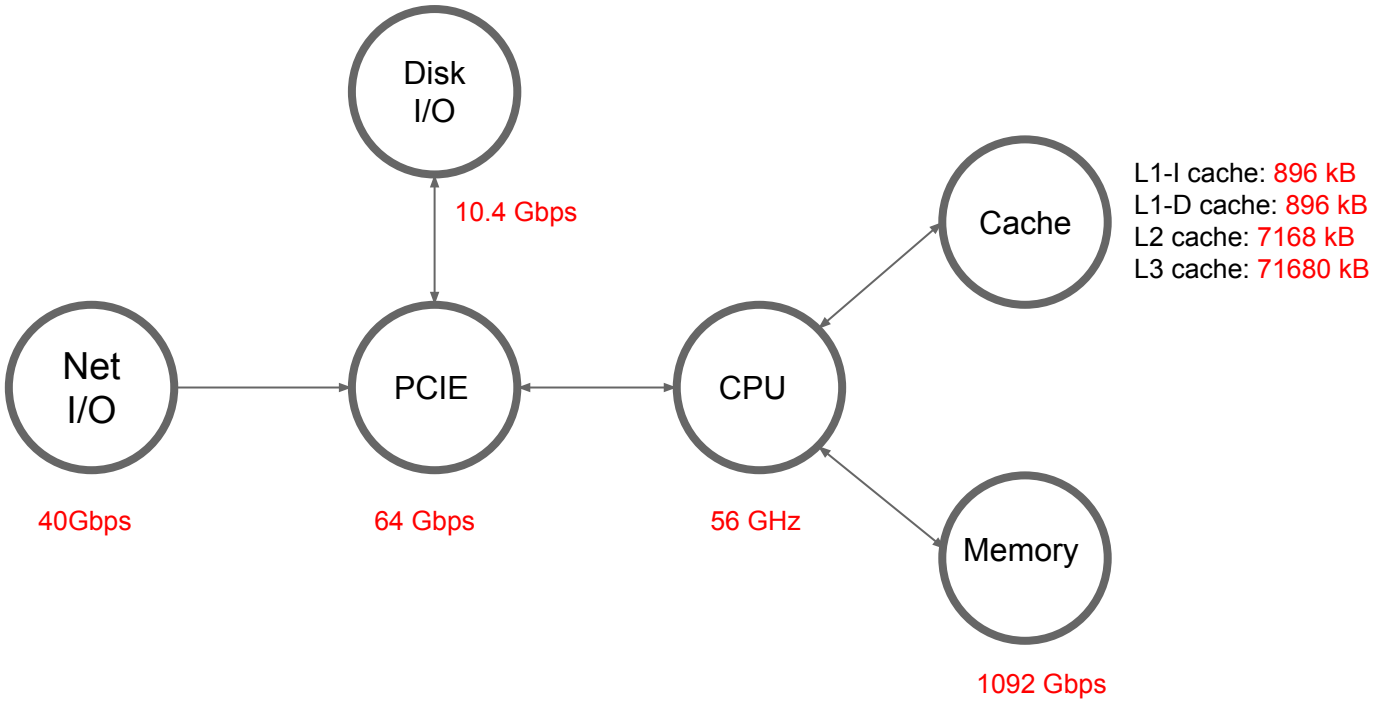


Shifting Micro-Bottlenecks

...or so.

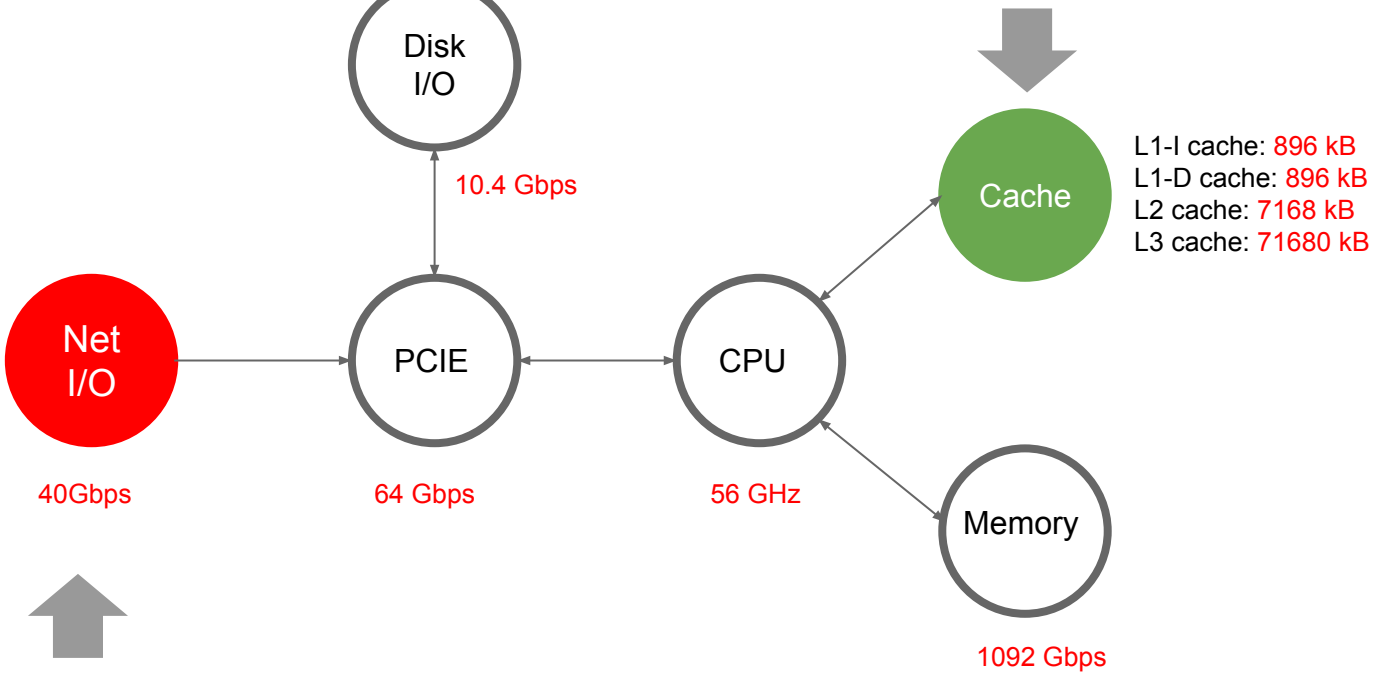


Non-linear Performance Collapse



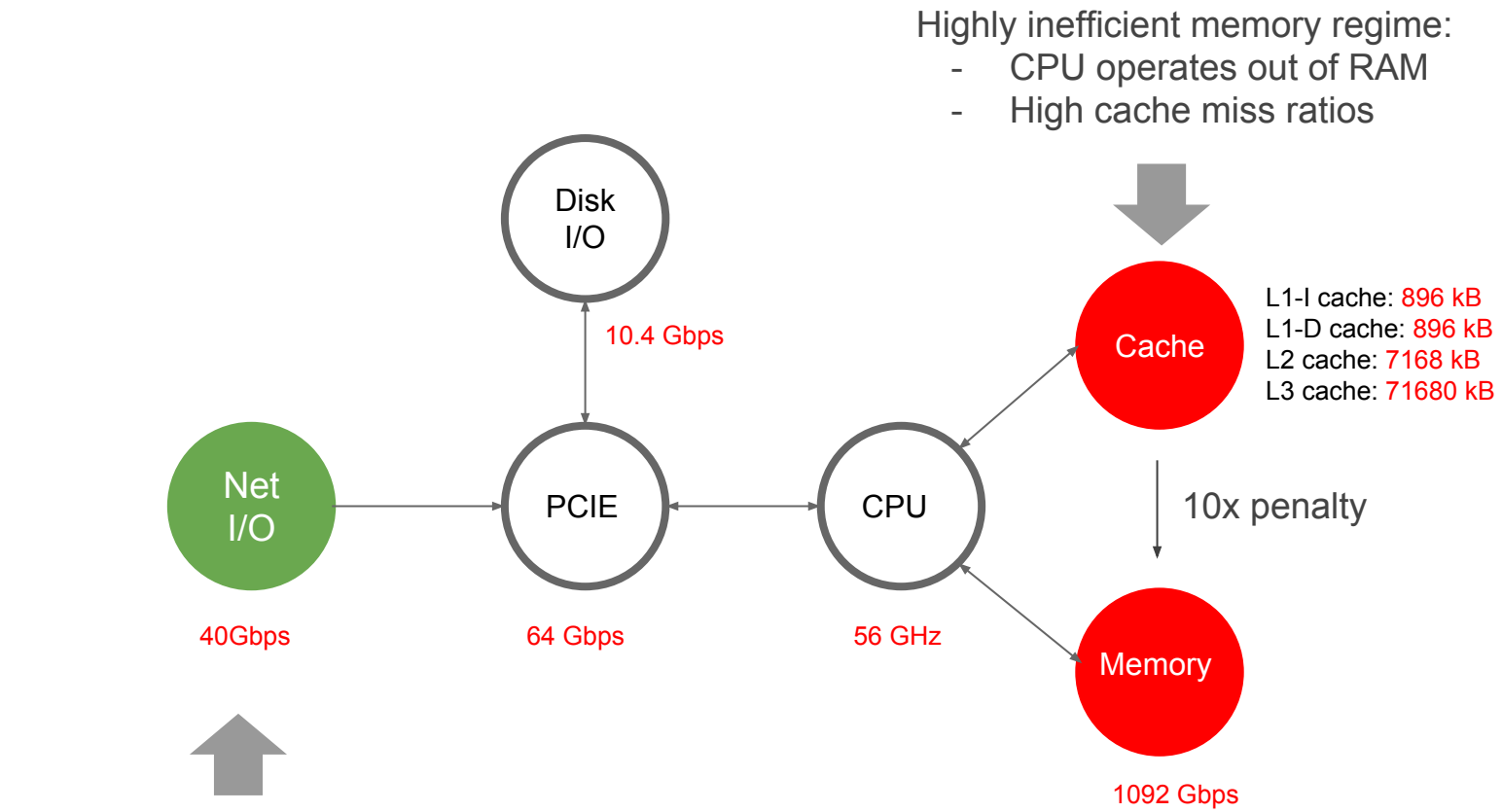
Non-linear Performance Collapse

Healthy cache regime:
- CPU operates out of cache
- High cache hit ratios



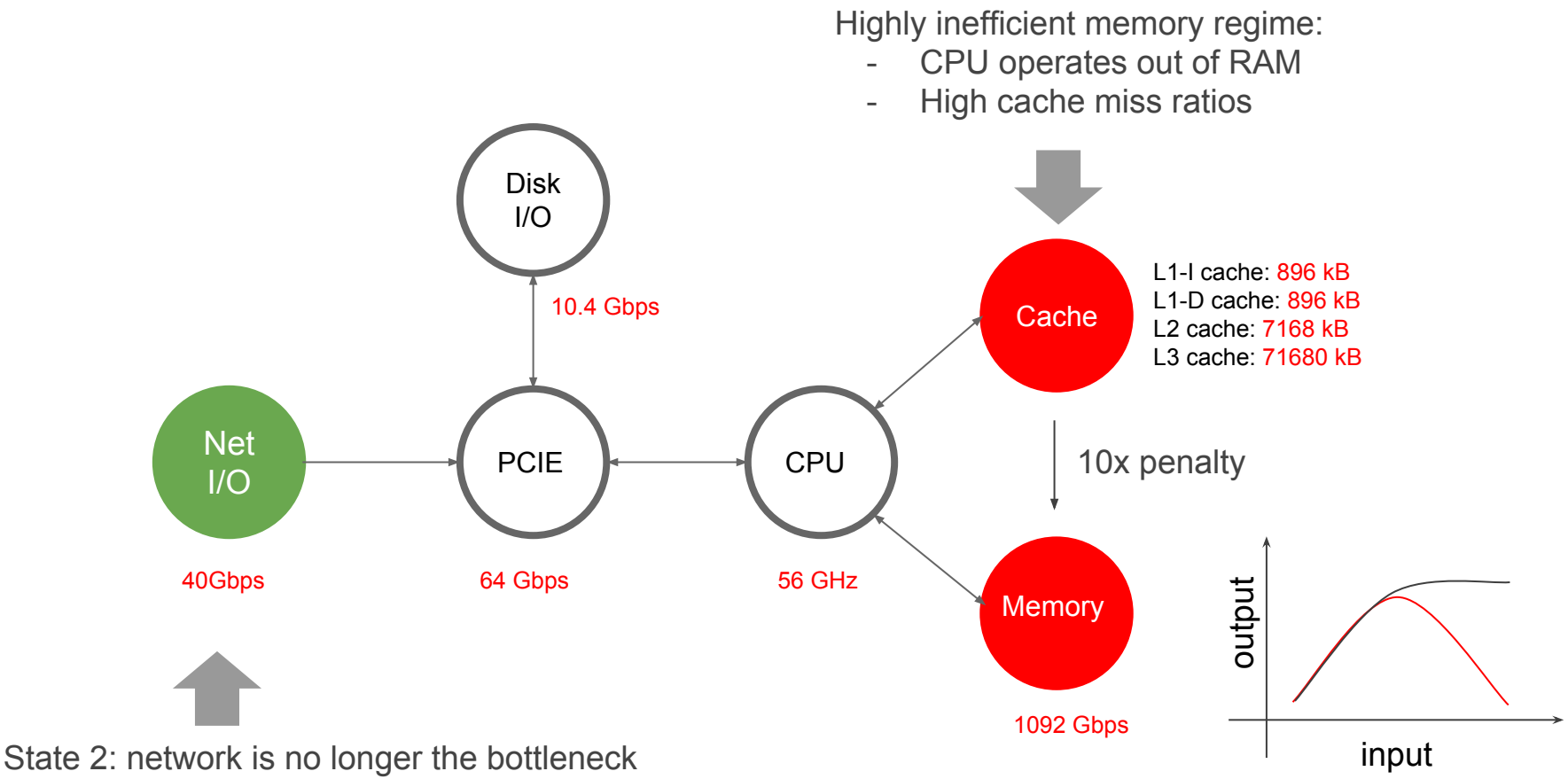
State 1: network is the bottleneck

Non-linear Performance Collapse



State 2: network is no longer the bottleneck

Non-linear Performance Collapse



By removing the network bottleneck, system spends more time processing packets that will need to be dropped anyway → net performance degradation (performance collapse)

Performance Optimization: Approach

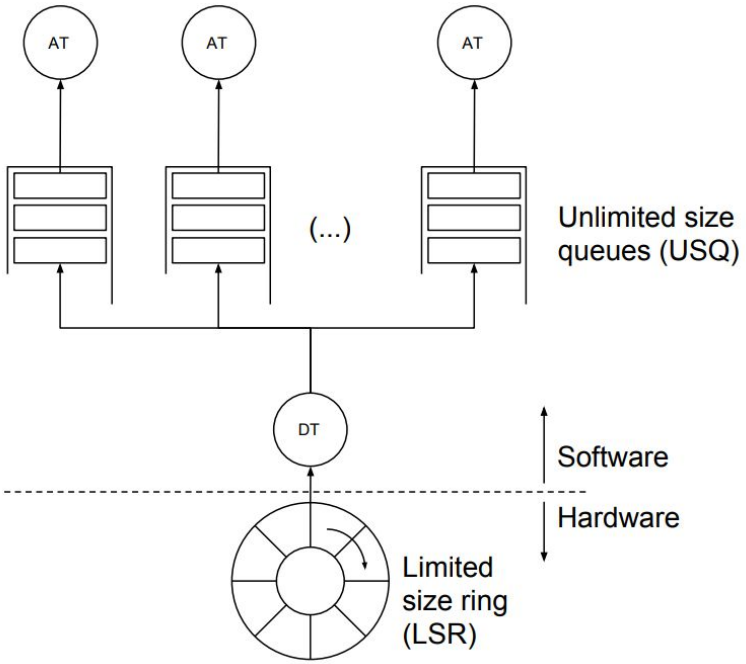
- The process of performance optimization needs to be a meticulous one involving small but safe steps to avoid the pitfall of pursuing short term gains that can lead to new and bigger bottlenecks down the path.

Performance Optimization: Algorithms and Data Structures

Long queue emulation	Reduces packet drops due to fixed-size hardware rings
Lockless bimodal queues	Improves packet capturing performance
Tail early dropping	Increases information entropy and extracted metadata
LFN tables	Reduces state sharing overhead
Multiresolution priority queues	Reduces cost of processing timers

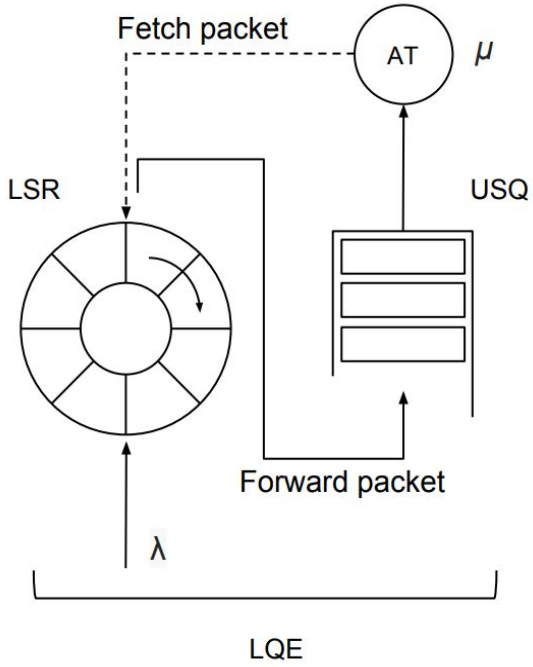
Long Queue Emulation

Dispatcher Model:



- Packet read cache penalty.
- Descriptor read cache penalty

Long queue emulation Model:



- Packet drop penalty under certain conditions

Long Queue Emulation

Lemma 1. Long queue emulation performance.

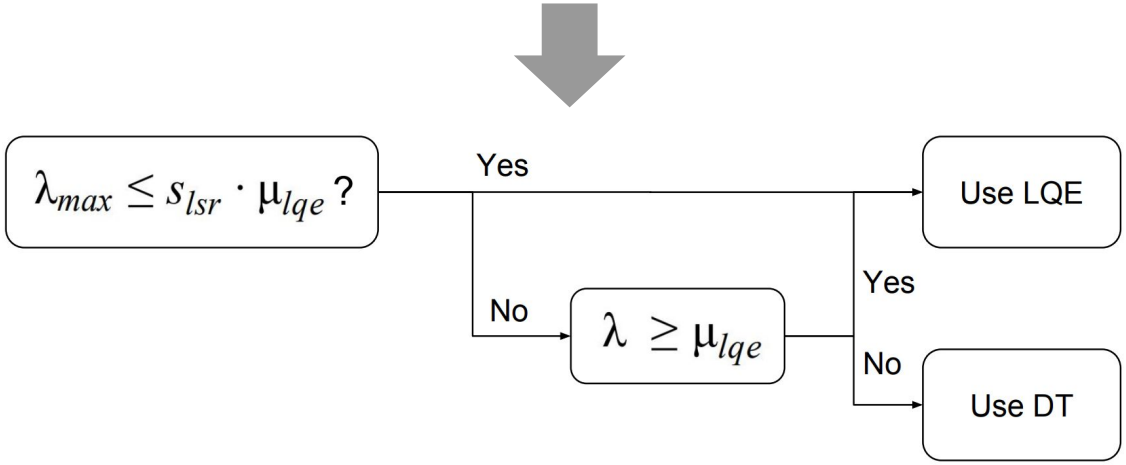
λ : average packet arrival rate

λ_{max} : maximum packet arrival rate

μ_{dt} : packet processing rate of the DT model

μ_{lqe} : packet processing rate of the LQE model

s_{lsr} : size of the LSR ring



Long Queue Emulation

Table 1. Maximum packet processing time for a Solarflare SFN7122F NIC

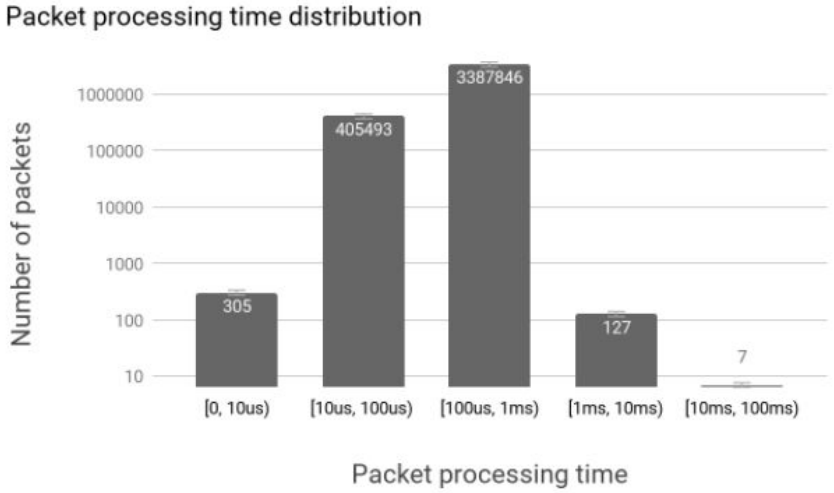
λ_{max} (Gbps)	1	2	4	6	8	10
s_{lsr}/λ_{max} (secs)	1.09	0.55	0.27	0.18	0.14	0.11

$\lambda_{max} \leq s_{lsr} \cdot \mu_{lqe}$?

Use LQE

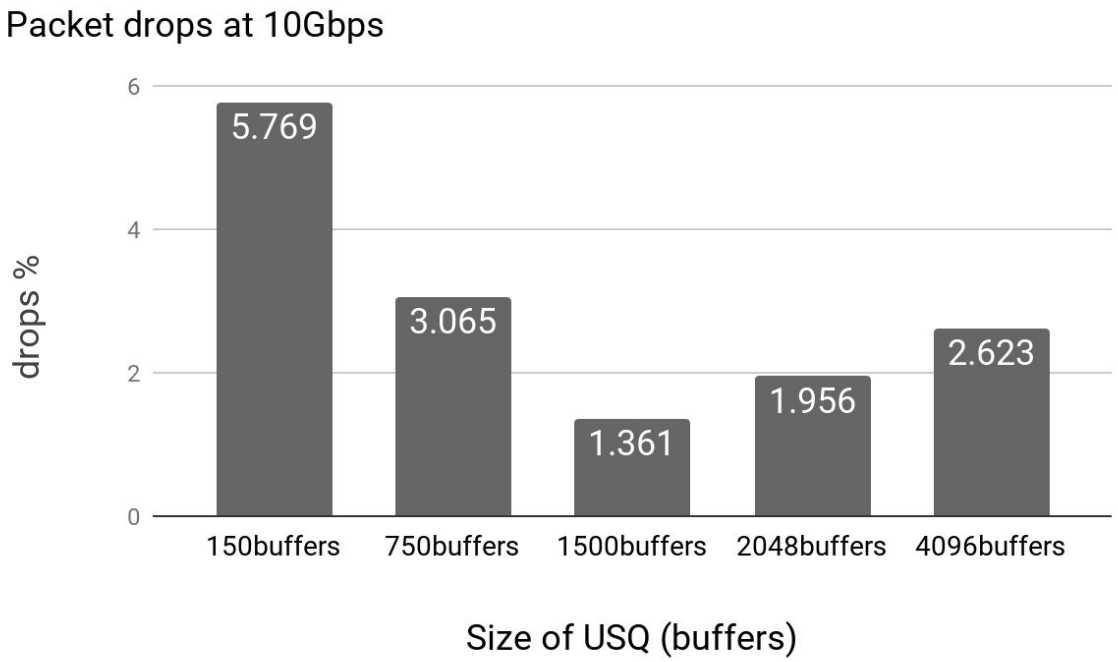
Table 2. *Packet processing time distribution.*

[0, 10us)	[10us, 100us)	[100us, 1ms)	[1ms, 10ms)	[10ms, 100ms)
305	405493	3387846	127	7
Total packets:	3793778			



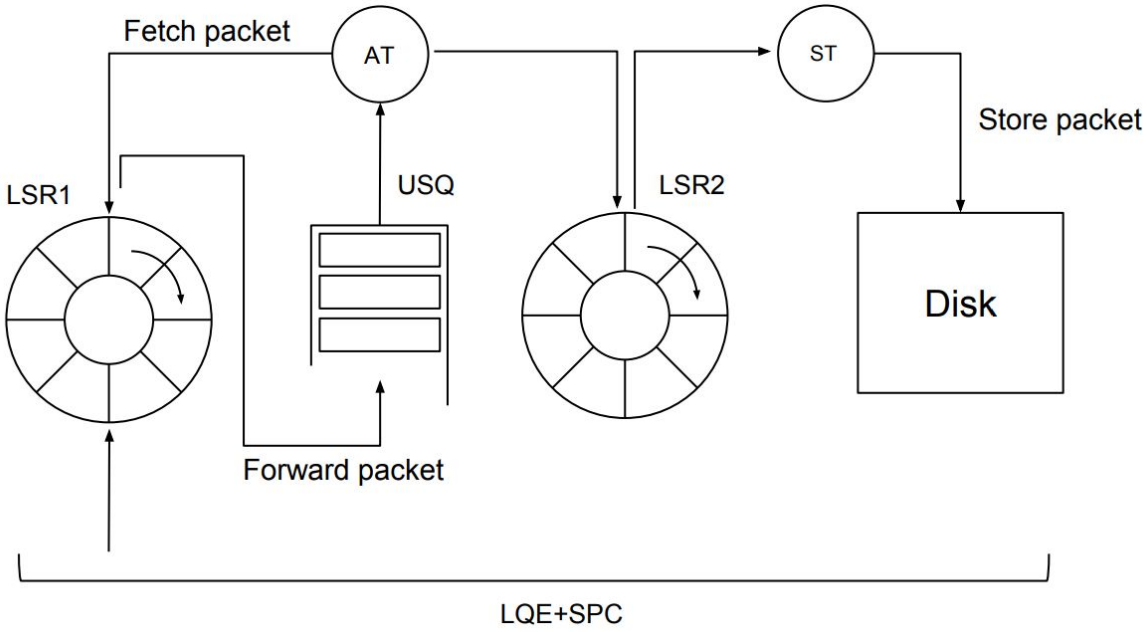
Long Queue Emulation

- Optimal LQE size



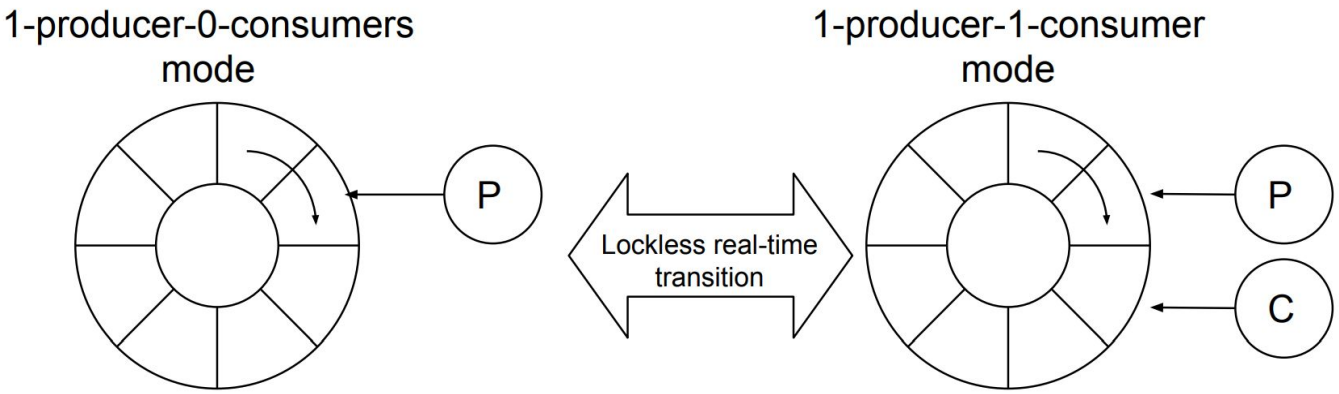
Lockless Bimodal Queues

- Goal: move packets from the memory ring to the disk without using locks



Lockless Bimodal Queues

- Goal: move packets from the memory ring to the disk without using locks



Lockless Bimodal Queues

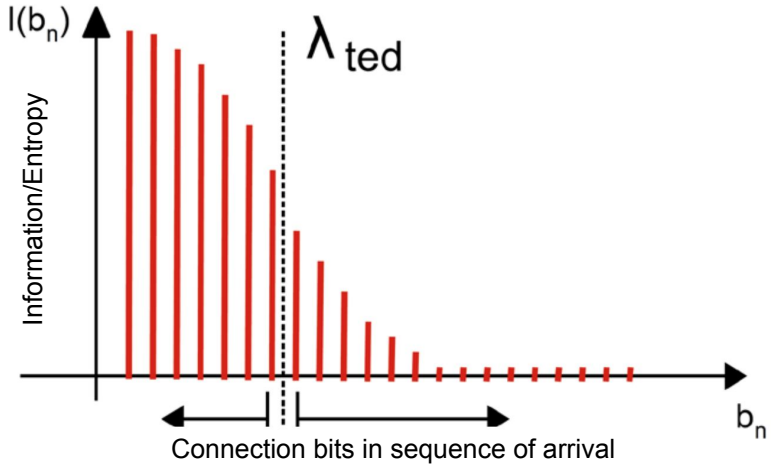
Lockless bimodal queue without using CAS
(producer must be permanently active to avoid consumer starvation)

```
1  typedef struct {
2    volatile unsigned int offset_p;
3    volatile unsigned int offset_c;
4    volatile bool req; // owned by consumer
5    volatile bool ack; // owned by producer
6    packet_t* vector[RINGSIZE];
7  } ring_t;
8
9  void enqueue(ring_t* ring, packet_t* pkt) {
10   if(!ring->req) {
11     if(ring->ack)
12       ring->ack = false;
13     if(ring->offset_p == ring->offset_c)
14       dequeue(ring);
15   }
16   else {
17       if(!ring->ack)
18         ring->ack = true;
19     while(ring->offset_p == ring->offset_c);
20   }
21   ring->vector[ring->offset_p++] = pkt;
22 }
23
24 packet_t* dequeue(ring_t* ring) {
25   if(ring->offset_p == ring->offset_c)
26     return NULL;
27   ring->offset_c = ring->offset_c + 1 % RINGSIZE;
28   return(vector[ring->offset_c - 1]);
29 }
30
31 void start_c(ring_t* ring) {
32   ring->req = true;
33   while(!ring->ack);
34 }
35
36 void stop_c(ring_t* ring) {
37   ring->req = false;
38   while(ring->ack);
39 }
```

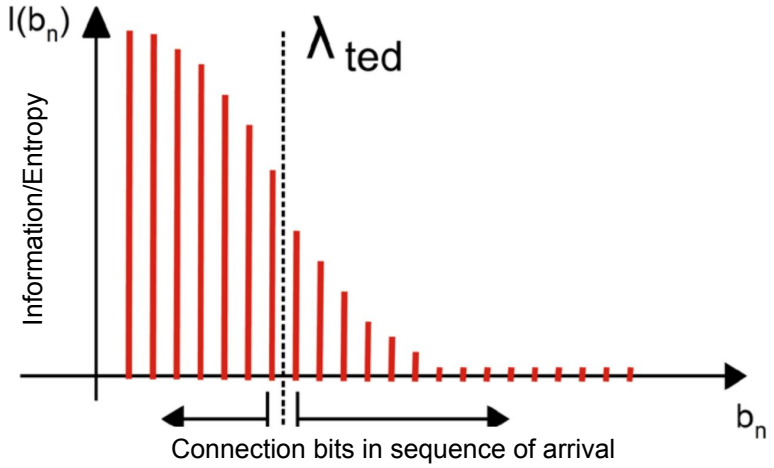
Lockless bimodal queue using CAS
(producer does not need to be permanently active)

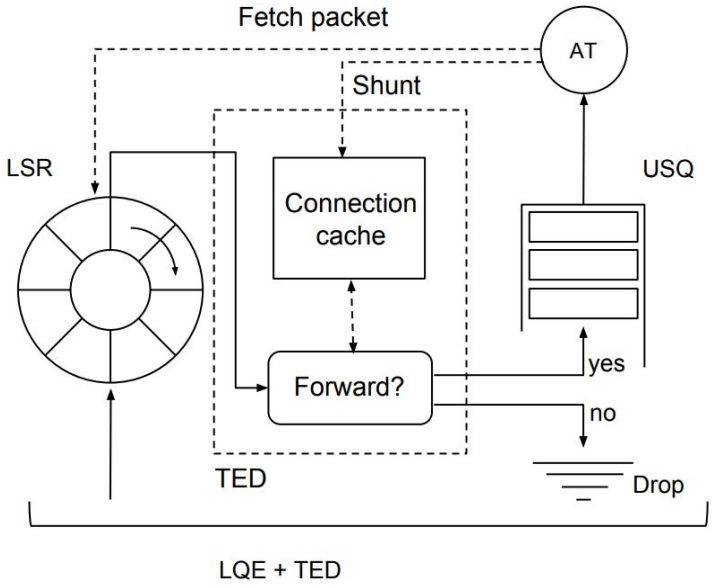
```
1  typedef struct {
2    volatile unsigned int offset_p;
3    volatile unsigned int offset_c;
4    volatile bool trans; // used to transition modes
5    volatile bool state; // the current mode
6    packet_t* vector[RINGSIZE];
7  } ring_t;
8
9  void enqueue(ring_t* ring, packet_t* pkt) {
10   while(!cas(&ring->lock, false, true));
11   if(!ring->state) {
12     if(ring->offset_p == ring->offset_c)
13       dequeue(ring);
14     else
15       while(ring->offset_p == ring->offset_c);
16   ring->trans = false;
17   ring->vector[ring->offset_p++] = pkt;
18 }
19 packet_t* dequeue(ring_t* ring) {
20   if(ring->offset_p == ring->offset_c) return NULL;
21   ring->offset_c = ring->offset_c + 1 % RINGSIZE;
22   return(ring->offset_c - 1);
23 }
24
25 void start_c(ring_t* ring) {
26   while(!cas(&ring->trans, false, true));
27   ring->state = true;
28   ring->trans = false;
29 }
30
31 void stop_c(ring_t* ring) {
32   while(!cas(&ring->trans, false, true));
33   ring->state = false;
34   ring->trans = false;
35 }
```

Tail Early Dropping



Tail Early Dropping

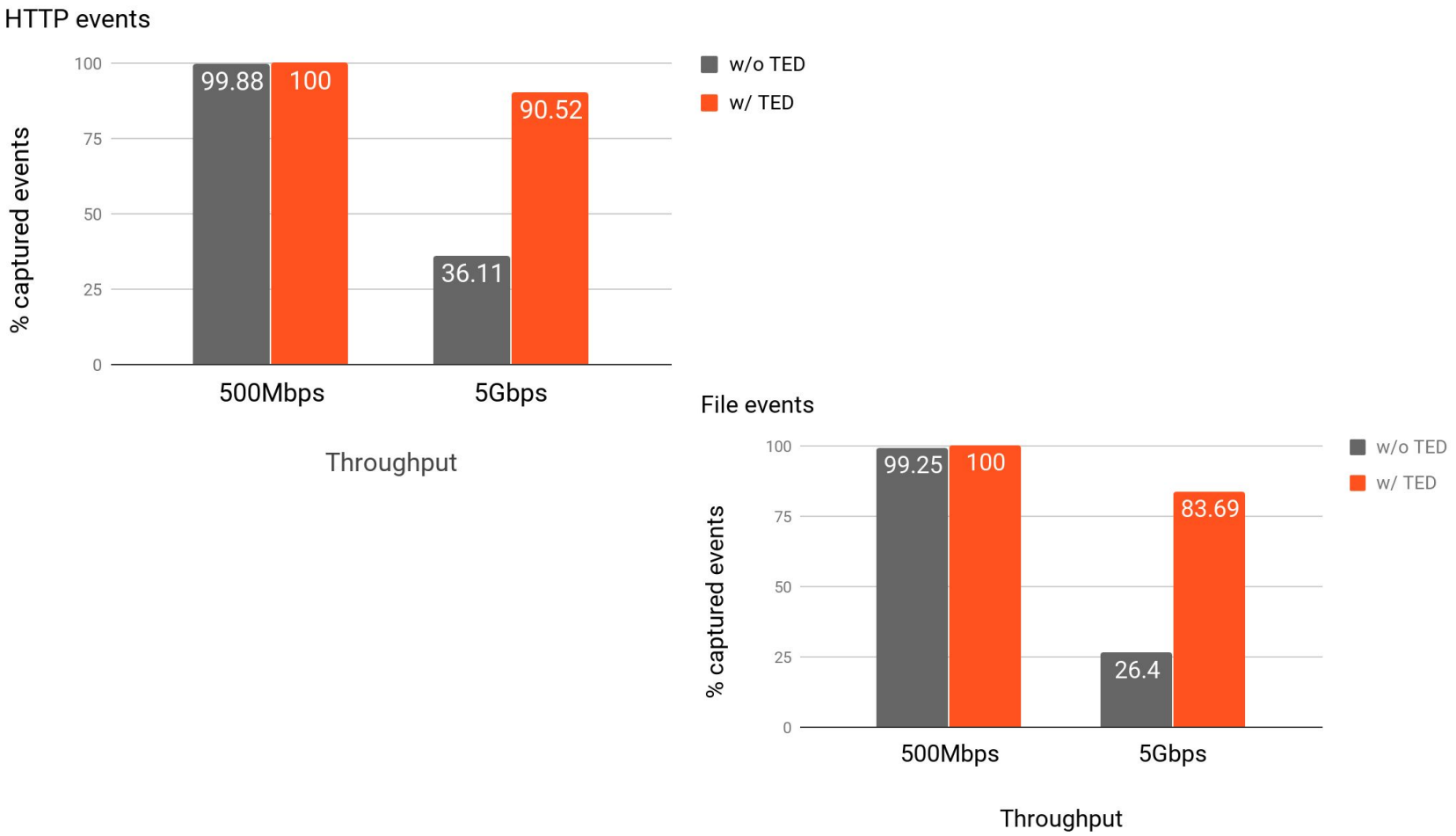




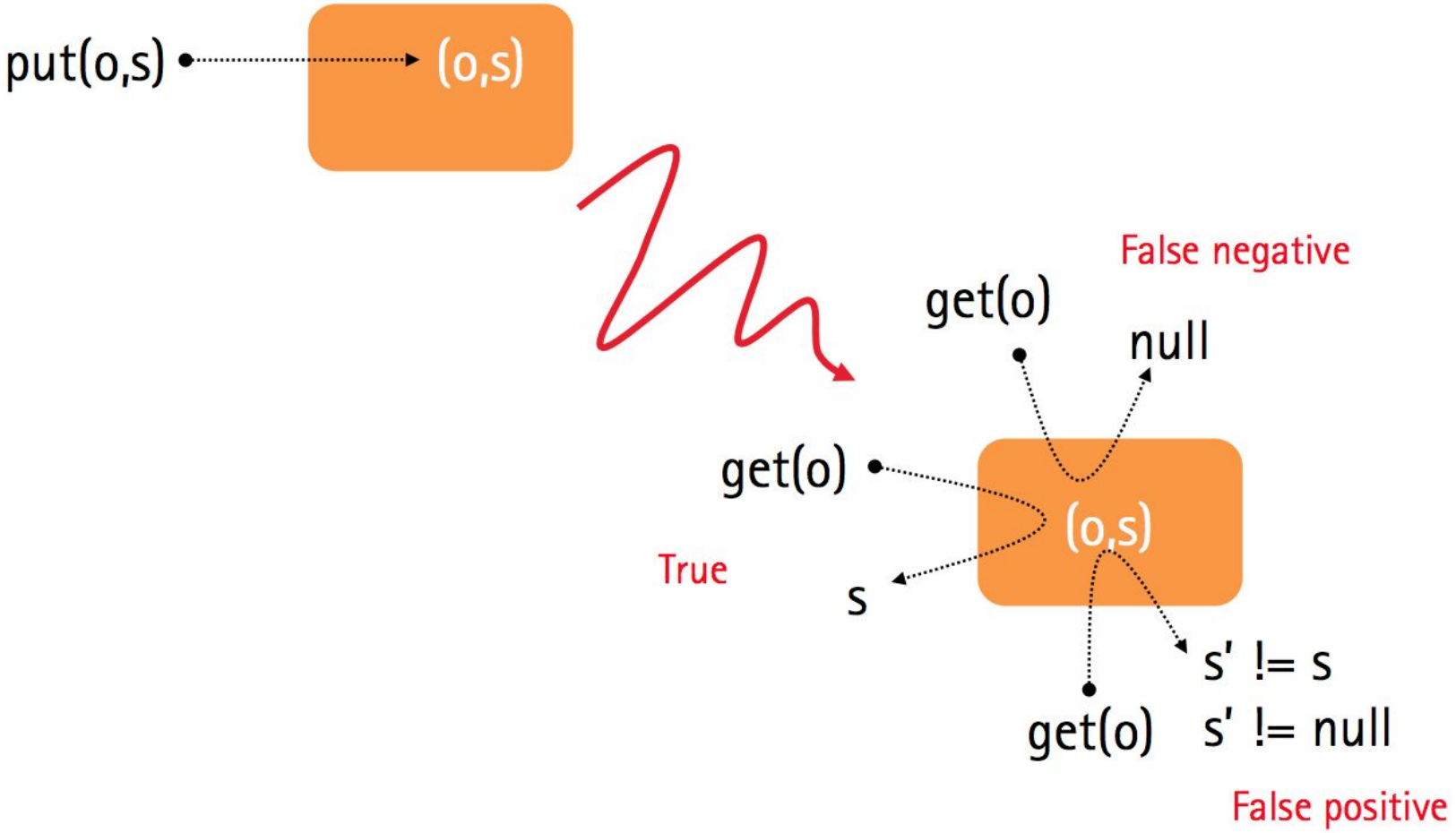
TED

```
1    Upon receiving a packet, do:
2      conn = lookup_connection_table(packet)
3      if conn.shunt or conn.packet_rec > ted_thr:
4        drop the packet
5      else:
6        forward the packet
7    Periodically, do:
8      if system is congested:
9        ted_thr = min(ted_thr / 2, ted_min);
10     else:
11       ted_thr += 1;
```

Tail Early Dropping



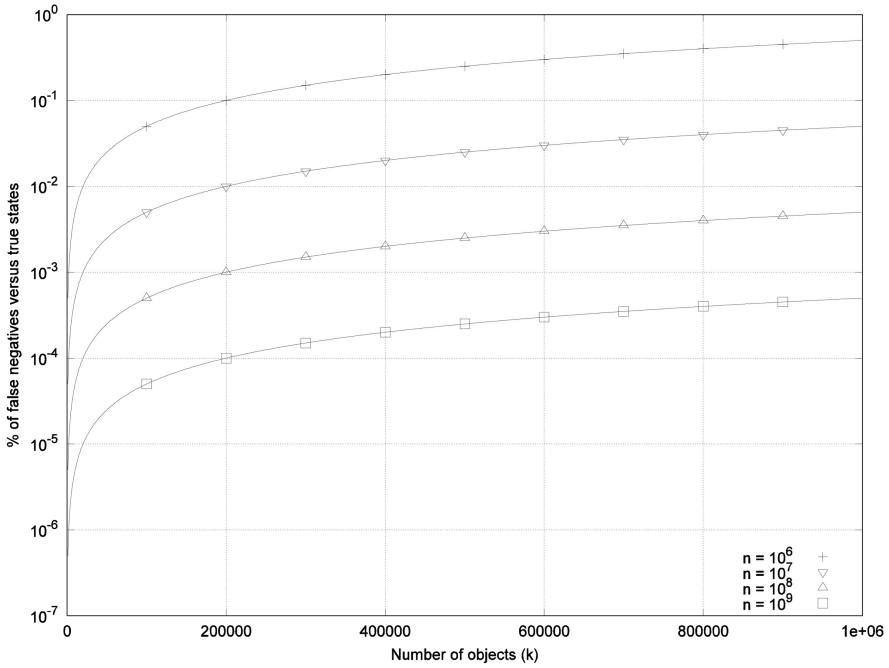
LFN Tables

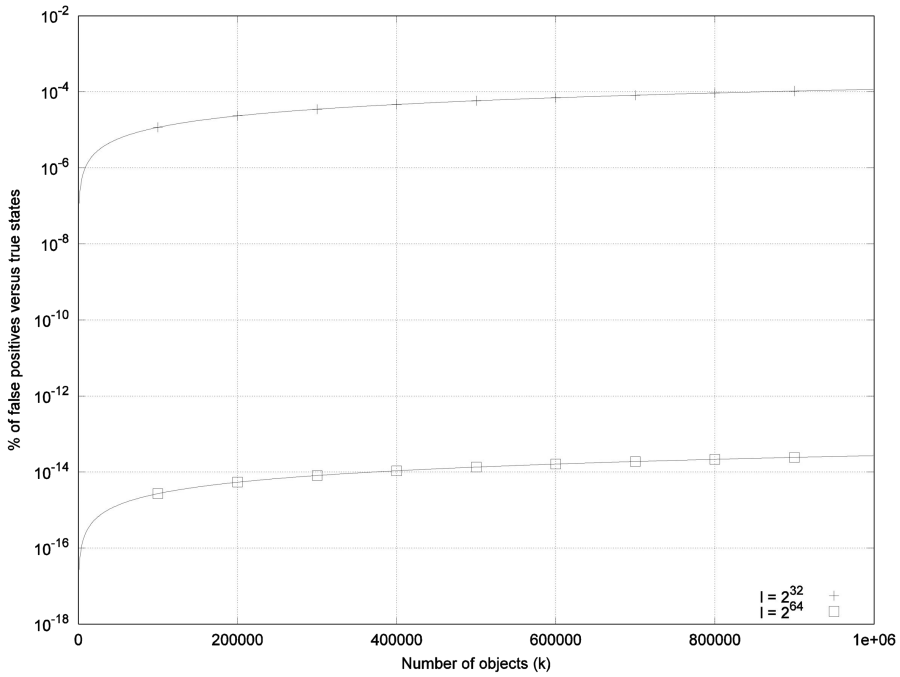

put(o,s)
(o,s)
False negative
get(o)
null
get(o)
(o,s)
True
s
s' != s
get(o)
s' != null
False positive

LFN Tables

```
Initial state:  T[e] = NULL for all e such that 0 ≤ e < n;
Parameters:
  n: size of the table
  l: processor's integer space size (typically 2^32 or 2^64)
h(x, k): the hash value of k modulo x
cat(x, y): concatenates the bytes from and x and y
put(k, v)
1   T[h(n, k)].value = v
2   T[h(n, k)].hash = h[l, cat(k, v)]
get(k)
3   if T[h(n, k)].hash == h(l, cat(k, T[h(n, k)].value)):
4     return T[h(n, k)].value
5   else:
6     return NULL
```

LFN Tables


% of false negatives versus true states
Number of objects (k)
n = 10^6
n = 10^7
n = 10^8
n = 10^9


% of false positives versus true states
Number of objects (k)
l = 2^32
l = 2^64

Multiresolution Priority Queues

- Priority queue: element at the front of the queue is the *greatest* of all the elements it contains, according to some total ordering defined by their *priority*.
- Found at the core of important computer science problems:
  - Shortest path problem
  - Packet scheduling in Internet routers
  - Event driven engines
  - Huffman compression codes
  - Operating systems
  - Bayesian spam filtering
  - Discrete optimization
  - Simulation of colliding particles
  - Artificial intelligence

Multiresolution Priority Queues

Year	Author	Data structure	Insert	Extract	Notes
1964	Williams [3]	Binary heap	O(log(n))	O(log(n))	Simple to implement.
1984	Fredman et al. [4]	Fibonacci Heaps	O(1)	O(log(n))	More complex to implement.
1988	Brown [8]	Calendar queues	O(1)	O(c)	Need to be balanced and resolution cannot be tuned.
2000	Chazelle [5]	Soft heaps	O(1)	O(1)	Unbounded error.
2008	Mehlhorn et al. [7]	Bucket queues	O(1)	O(c)	Priorities must be small integers and resolution cannot be tuned.
2017	Ros-Giralt et al. (this work)	Multiresolution priority queue	O(1), O(r) or O(log(r))	O(1)	Tunable/bounded resolution error. Error is zero if priority space is multi-resolutive.

n: number of elements in the queue
c: maximum integer priority value
r: number of resolution groups supported by the multiresolution priority queue

Multiresolution Priority Queues

- A multiresolution priority queue is a container data structure that at all times maintains the following invariant:

Property 1. Multiresolution Priority Queue (MR-PQ) Invariant. Let e_i and e_j be two arbitrary elements with priorities p_i and p_j, respectively, where $p_{min} \leq p_i < p_{max}$ and $p_{min} \leq p_j < p_{max}$. Then for all possible states, a multiresolution priority queue ensures that element e_i is dequeued before element e_j if the following condition is true:

$$\lfloor (p_i - p_{min})/p_\Delta \rfloor < \lfloor (p_j - p_{min})/p_\Delta \rfloor \quad (1)$$

§

- Intuitively:
  1. Discretize the priority space into a sequence of slots or resolution groups $[p_{min}, p_{min} + p_\Delta)$, $[p_{min} + p_\Delta, p_{min} + 2 \cdot p_\Delta)$, …, $[p_{max} - p_\Delta, p_{max})$
  2. Prioritize elements according to the slot in which they belong.
  3. Elements belonging to lower slots are given higher priority.
  4. Within a slot, ordering is not guaranteed. This enables a mechanism to control the trade-off accuracy versus performance.

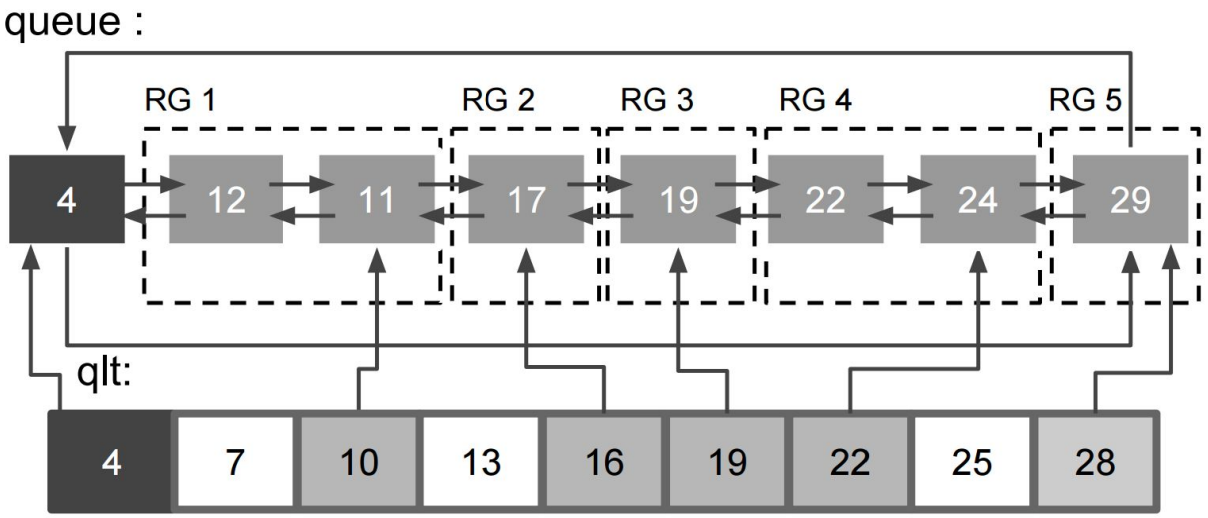
Multiresolution Priority Queues

- The larger the parameter p_Δ → the lower the resolution of the queue → the higher the error → the higher the performance (and vice versa)
- Instead of ordering the space of elements, an MR-PQ orders the space of priorities.
- The information theoretic barriers of the problem are broken by introducing error in a way that entropy is reduced:
  - In many real world problems, the space of priorities has much lower entropy than the space of keys.
  - Example:
    - Space of keys is the set of real numbers (S_k)
    - Space of priorities is the set of distances between any two US cities (S_p)
    - Entropy(S_k) >> Entropy(S_p)

Multiresolution Priority Queues

- How it works through an example.

Let a multiresolution priority queue have parameters $p_{\Delta} = 3$, $p_{min} = 7$ and $p_{max} = 31$, and assume we insert seven elements with priorities 19, 11, 17, 12, 24, 22 and 29 (inserted in this order). Then:



Multiresolution Priority Queues: Base Algorithm

BUILD(q)

```
1    sentinel = alloc_element();
2    queue = sentinel;
3    queue->next = queue;
4    queue->prev = queue;
5    qltsize = (pmax-pmin)/pdelta + 1;
6    for i in [1, qltsize):
7        qlt[i] = NULL;
8    qlt[0] = queue;
9    queue->prio = pmin - pdelta;
```

Multiresolution Priority Queues: Base Algorithm

QLTSLOT(e)

```
24   slot = (int)((e->prio - queue->prio)/pdelta);
25   return slot;
```

INSERT(e)

```
10   slot = slot_iter = QLTSLOT(e);
11   while qlt[slot_iter] == NULL:
12       slot_iter--;
13   if slot_iter == slot: // Add to the left
14       e->next = qlt[slot];
15       e->prev = qlt[slot]->prev;
16       qlt[slot]->prev->next = e;
17       qlt[slot]->prev = e;
18   else:  // Add to the right
19       e->next = qlt[slot_iter]->next;
20       e->prev = qlt[slot_iter];
21       qlt[slot_iter]->next->prev = e;
22       qlt[slot_iter]->next = e;
23       qlt[slot] = e;
```

Multiresolution Priority Queues: Base Algorithm

QLTREPAIR(e)

```
34   slot = QLTSLOT(e);
35   if qlt[slot] != e:
36       return; // Nothing to fix
37   if slot == QLTSLOT(e->prev):
38       qlt[slot] = e->prev; // Fix the slot
39   else:
40       qlt[slot] = NULL; // No elements left in
     slot
```

EXTRACT(e)

```
30   e->prev->next = e->next;
31   e->next->prev = e->prev;
32   QLTREPAIR(e);
33   return e;
```

Multiresolution Priority Queues: Base Algorithm

PEEK()

```
26   e = q->next;
27   if e == q:
28       return NULL;  // Queue is empty
29   return e;
```

EXTRACTMIN()

```
41   e = PEEK();
42   EXTRACT(e);
43   return e;
```

Multiresolution Priority Queues: Correctness and Complexity

Lemma 1. Correctness of the MR-PQ algorithm. The INSERT and REMOVE routines preserve the MR-PQ invariant (Property 1).

Lemma 2. Complexity of the MR-PQ algorithm. The worst case complexity of the MR-PQ algorithm for the INSERT routine is $O(r)$, where $r = (p_{max} - p_{min})/p_{\Delta}$ is the number of resolution groups supported by the queue. The complexity of the INSERT routine becomes $O(1)$ if there is at least one element in each slot. The complexity of the PEEK, EXTRACTMIN and EXTRACT routines is $O(1)$.

Multi-resolutive Priority Spaces

Definition. Multi-resolutive priority space. Let $\mathcal{P}$ be the set of possible priorities that the elements in a priority queue can take and assume that $\{p_1(t), p_2(t), ..., p_n(t)\}$ is the set of priorities of the elements stored in the queue at an arbitrary time t. We will say that $\mathcal{P}$ is multi-resolutive with resolution r if there exists a set $\{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_r\}$ such that the following three conditions are true

(1) $\bigcup_{i=1}^{r} \mathcal{P}_i = \mathcal{P}$ and $\mathcal{P}_i \cap \mathcal{P}_j = \varnothing$ for $i \neq j$ (i.e., $\{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_r\}$ is an r-part partition of $\mathcal{P}$)

(2) if $p_i(t) \in \mathcal{P}_k$, then $p_j(t) \notin \mathcal{P}_k$, $\forall j \neq i$ and $\forall t$

(3) and r is minimal.

- Example:

Consider the priority set of real numbers, $\mathcal{P} = \mathbb{R}$ and assume that at any arbitrary time, any two elements e_i and e_j in the queue are guaranteed to not have priorities closer than a nor further than b: $|p_i - p_j| > a$ and $|p_i - p_j| < b$. Then we can say that $\mathcal{P}$ is multi-resolutive with resolution $r = \lceil b/a \rceil$.

Multi-resolutive Priority Spaces

- Problems with multi-resolutive priority spaces can be resolved by a multi-resolution priority queue at a faster speed and without adding any additional error. In this case, we achieve better performance at no cost.
- If condition (2) does not hold, then an error is introduced but the entropy of the problem stays constant. In this case, we also achieve better performance but at the cost of losing some accuracy.

Multi-resolutive Priority Spaces

- We can apply the rules of multi-resolutive priority spaces to optimize the performance of problems involving priority queues.
- Example. Consider the classic shortest path problem, known to have a complexity of O((v+e)log(v)), for a graph with v vertices and e edges (See Section 24.3 of [2]). By using a multiresolution priority queue we have:
  - If the graph is such that the edge weights define a multi-resolutive priority space, then using MR-PQ we can find the exact shortest path with a cost O(v+e).
  - Otherwise, we can find the *approximate* shortest path with a cost O(v+e) and with a controllable error given by the parameter *r*.

Multiresolution Priority Queues: Support for Sliding Priorities

- The base MR-PQ algorithm assumes priorities are in the set [pmin, pmax).
- Data structure can be generalized to support priorities in the set $[p_{min}+d(t), p_{max}+d(t))$, where d(t) is any monotonically increasing function of a parameter t.
- The case of sliding priority sets is particularly relevant to applications that run event-driven engines. (See for example Section 6.5 of [2].)

Multiresolution Priority Queues: Support for Sliding Priorities

- Sliding priorities can be supported with a few additional lines of code:

QLTSLIDE(prio)

```
44   shift = (prio - queue->prio)/pdelta - 1;
45   if shift < 1:
46       return;
47   for i in [1, qltsize):
48       if i < qltsize - shift:
49           qlt[i] = qlt[i + shift];
50       else
51           qlt[i] = 0;
52   queue->prio = prio - pdelta;
```

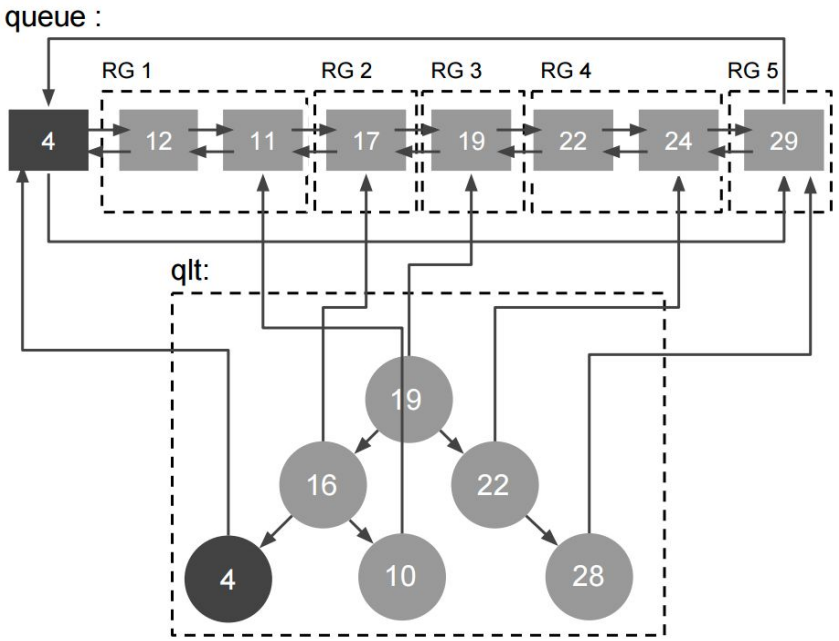
EXTRACTMIN()

```
53   e = PEEK();
54   EXTRACT(e);
55   QLTSLIDE(e->prio);    // Added to support sliding
                           // priorities
56   return e;
```

Lemma 3. Correctness of the MR-PQ algorithm with sliding priorities. The modified EXTRACTMIN routine preserves the MR-PQ invariant (Property 1).

Multiresolution Priority Queues: Binary Heap Based Optimization

- When not all the slots in the QLT table are filled in, performance can be improved by implementing the QLT table using a binary heap.
- Let a multiresolution priority queue have parameters $p_{\Delta} = 3$, $p_{min} = 7$ and $p_{max} = 31$, and assume we insert seven elements with priorities 19, 11, 17, 12, 24, 22 and 29 (inserted in this order). Then:



Multiresolution Priority Queues: Complexity Analysis

Table 1. Computational cost.

Algorithm	INSERT	PEEK	EXTRACTMIN	EXTRACT
BH-PQ	$log(n)$	$O(1)$	$log(n)$	$log(n)$
MR-PQ	$O(r)$ or $O(1)$	$O(1)$	$O(1)$	$O(1)$
BT-MR-PQ	$O(log(r))$	$O(1)$	$O(1)$	$O(1)$

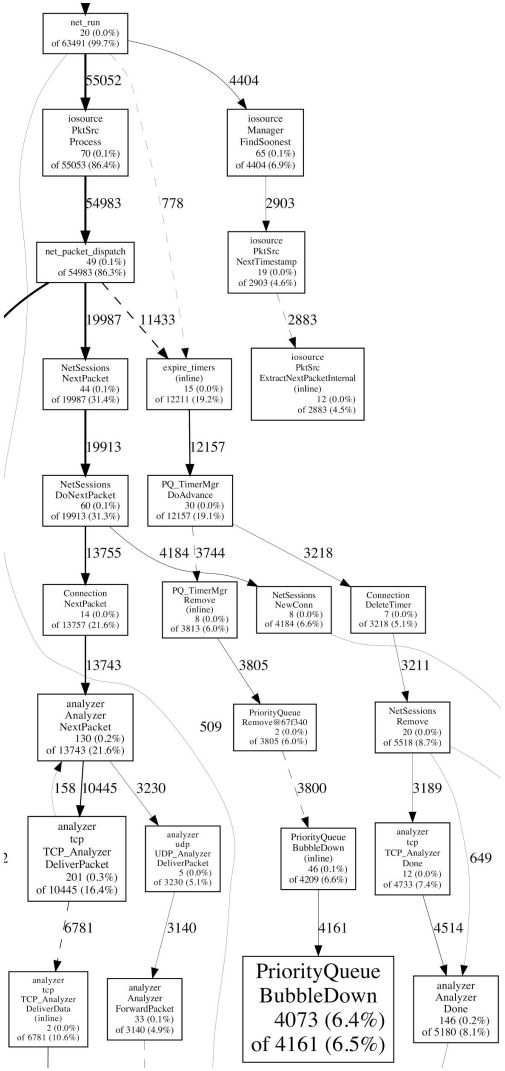
Multiresolution Priority Queues: Benchmarks

- We use MR-PQ to resolve a real world HPC problem.
- Problem statement: when running the Bro network analyzer [12] against very high speed traffic consisting of many short lived connections, the BubbleDown operation in the (binary heap based) priority queue used to manage Bro timers becomes a main system bottleneck.
- Top functions in Bro according to their computational cost:

```
Total: 63724 samples
    4139   6.5% PriorityQueue::BubbleDown
    2500   3.9% SLL_Pop
    1899   3.0% Ref
    1829   2.9% Unref
    1701   2.7% PackedCache::KeyMatch
    1537   2.4% Attributes::FindAttr
    1249   2.0% Dictionary::Lookup
    1184   1.9% NameExpr::Eval
```

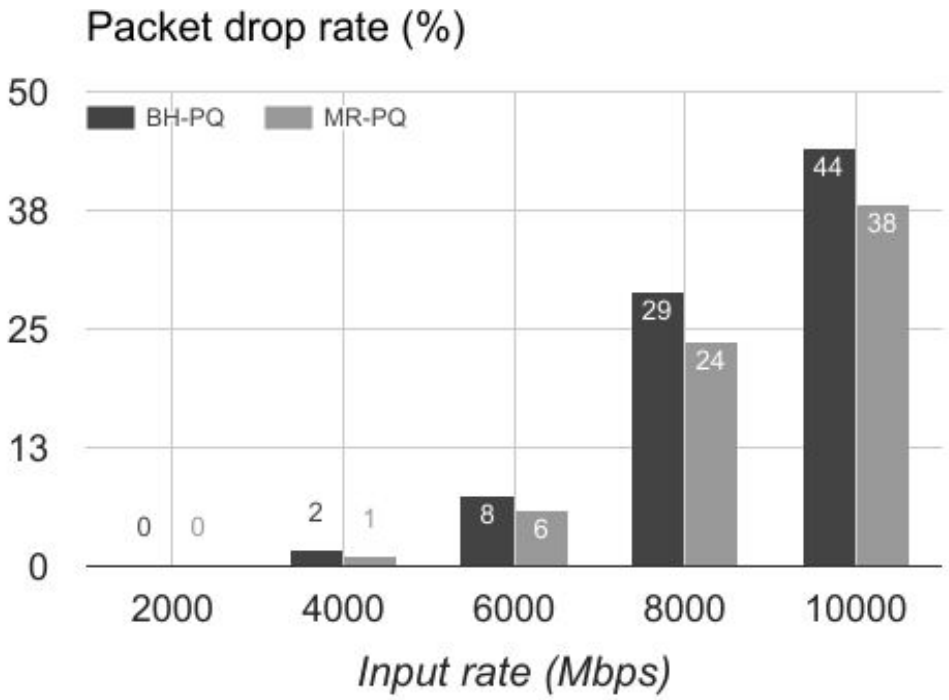
Multiresolution Priority Queues: Benchmarks

- Call graph showing the BubbleDown function as the main bottleneck



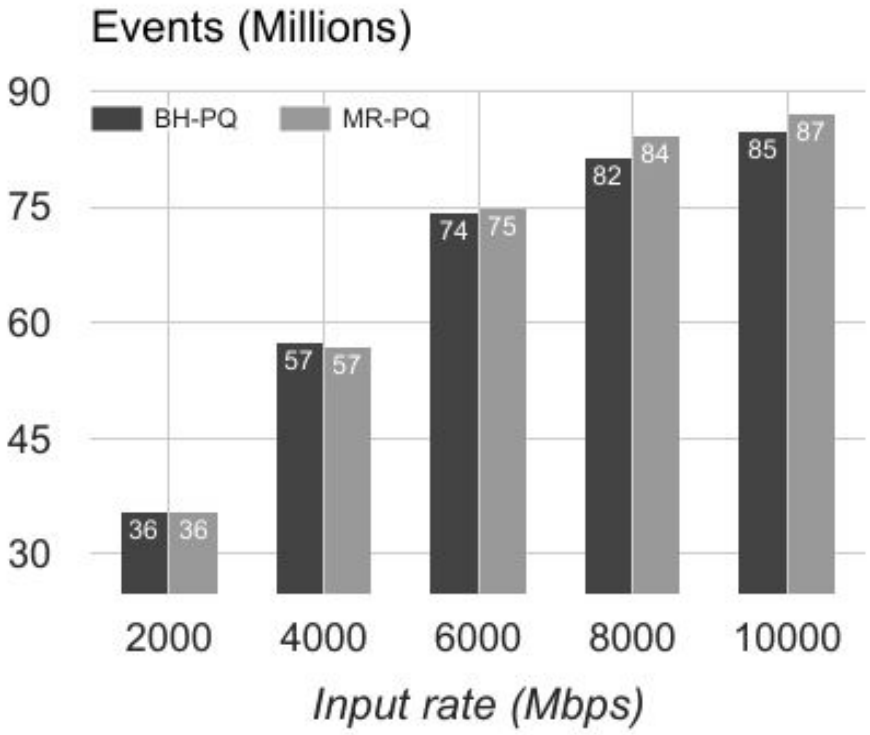
Multiresolution Priority Queues: Benchmarks

BH-PQ: Bro with its standard binary heap based priority queue to manage timers
MR-PQ: Bro using a multiresolution priority queue to manage timers



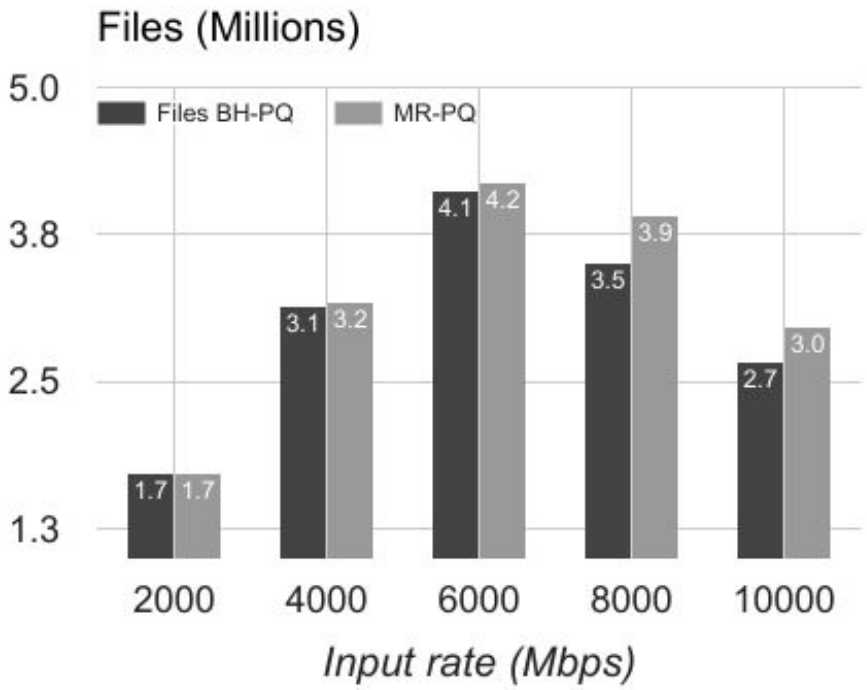
Multiresolution Priority Queues: Benchmarks

BH-PQ: Bro with its standard binary heap based priority queue to manage timers
MR-PQ: Bro using a multiresolution priority queue to manage timers



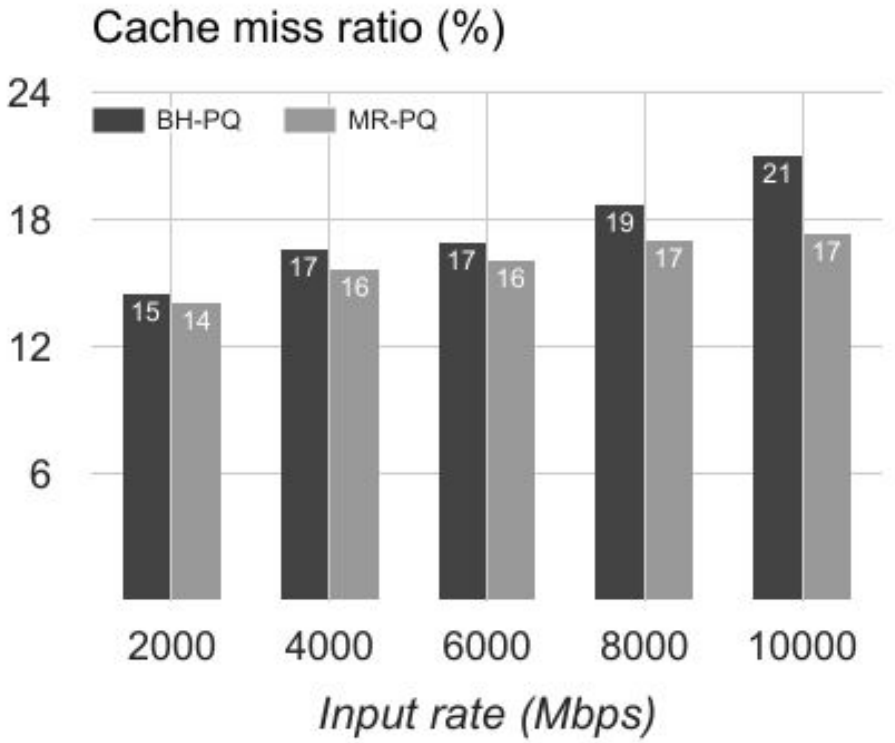
Multiresolution Priority Queues: Benchmarks

BH-PQ: Bro with its standard binary heap based priority queue to manage timers
MR-PQ: Bro using a multiresolution priority queue to manage timers



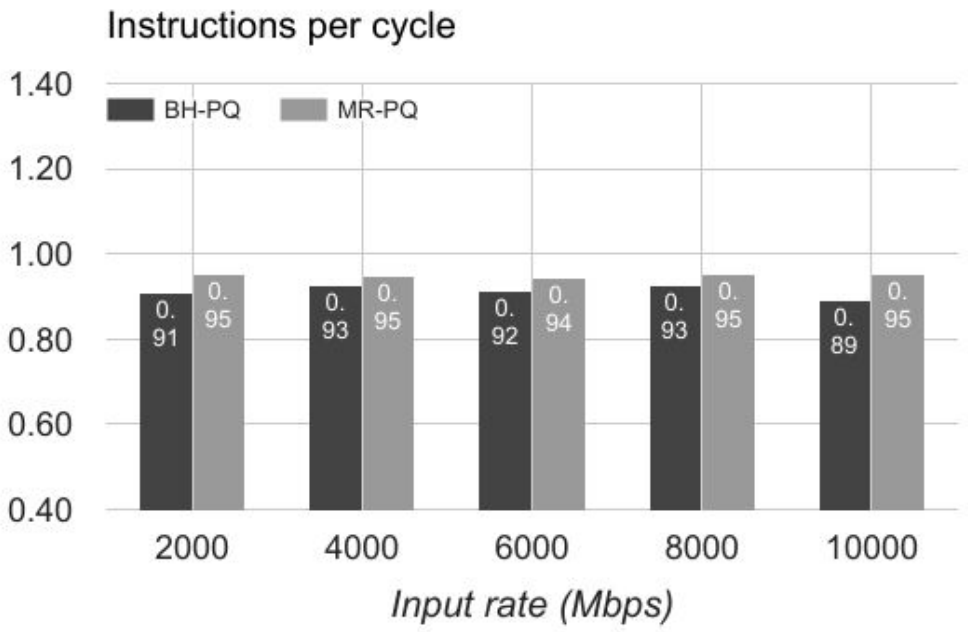
Multiresolution Priority Queues: Benchmarks

BH-PQ: Bro with its standard binary heap based priority queue to manage timers
MR-PQ: Bro using a multiresolution priority queue to manage timers

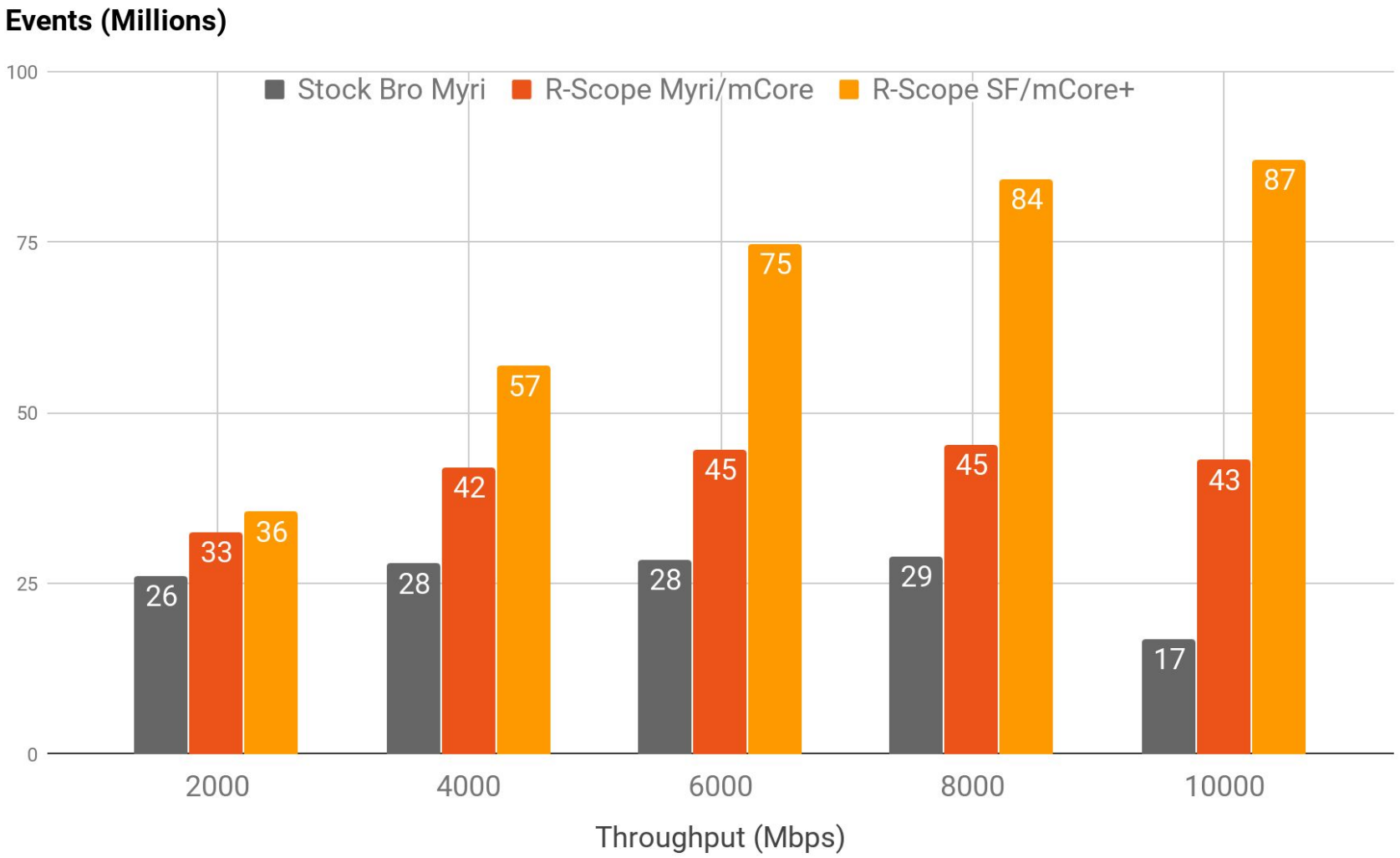


Multiresolution Priority Queues: Benchmarks

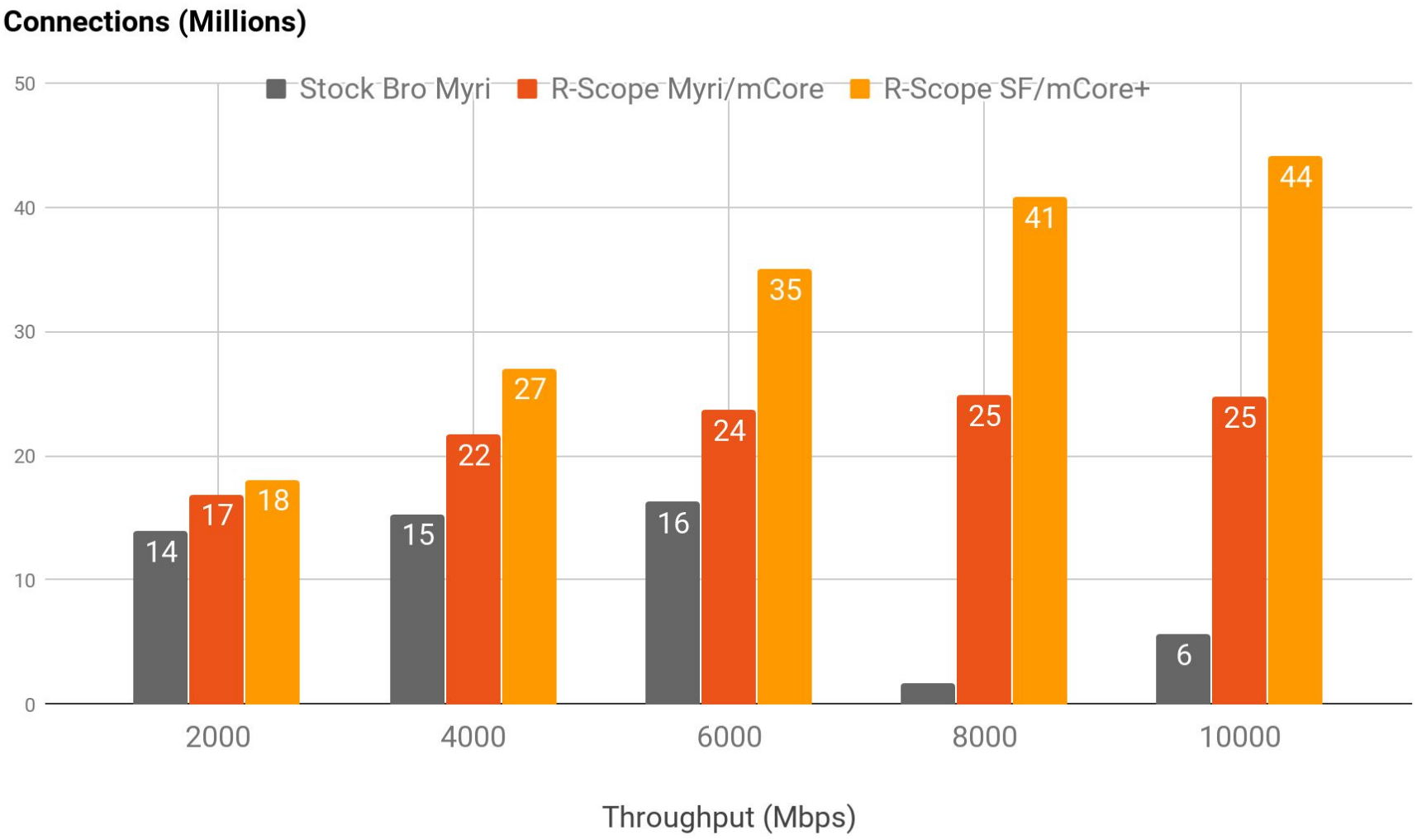
BH-PQ: Bro with its standard binary heap based priority queue to manage timers
MR-PQ: Bro using a multiresolution priority queue to manage timers



R-Scope Network Security Sensor: System Wide Benchmarks



R-Scope Network Security Sensor: System Wide Benchmarks

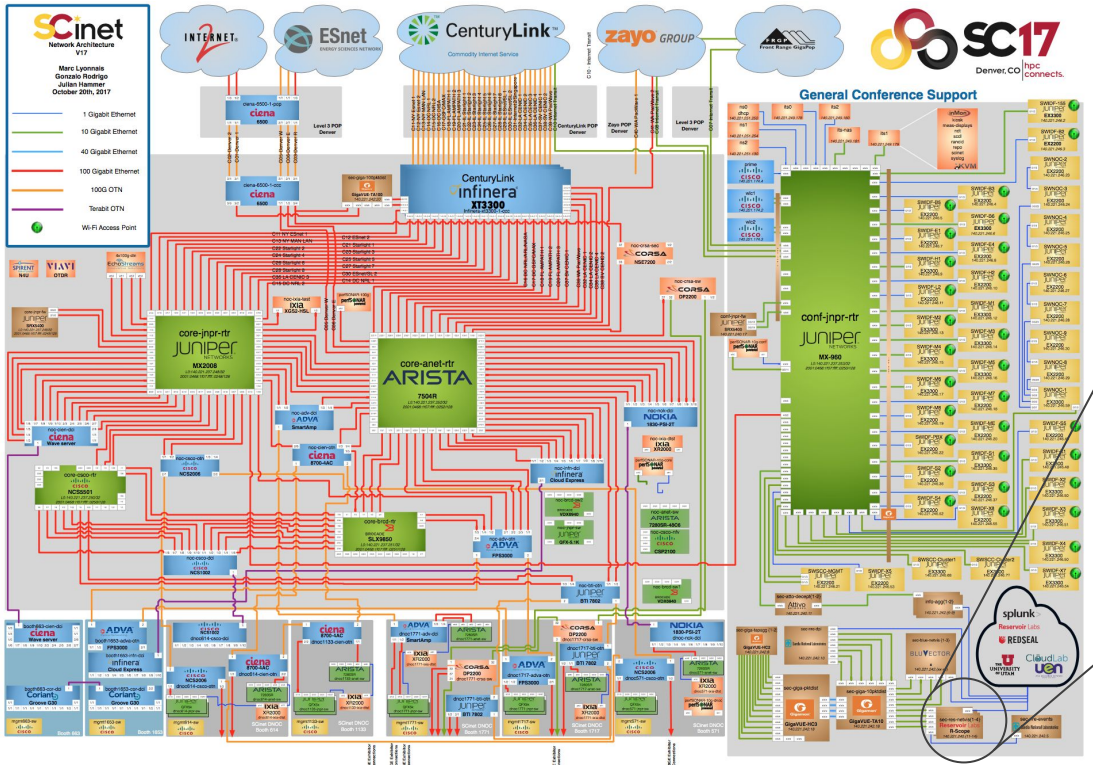


R-Scope Providing Deep Network Visibility at SC2017 / SCinet

DENVER TO BECOME EPICENTER FOR FASTEST INTERNET AND COMPUTERS IN THE WORLD

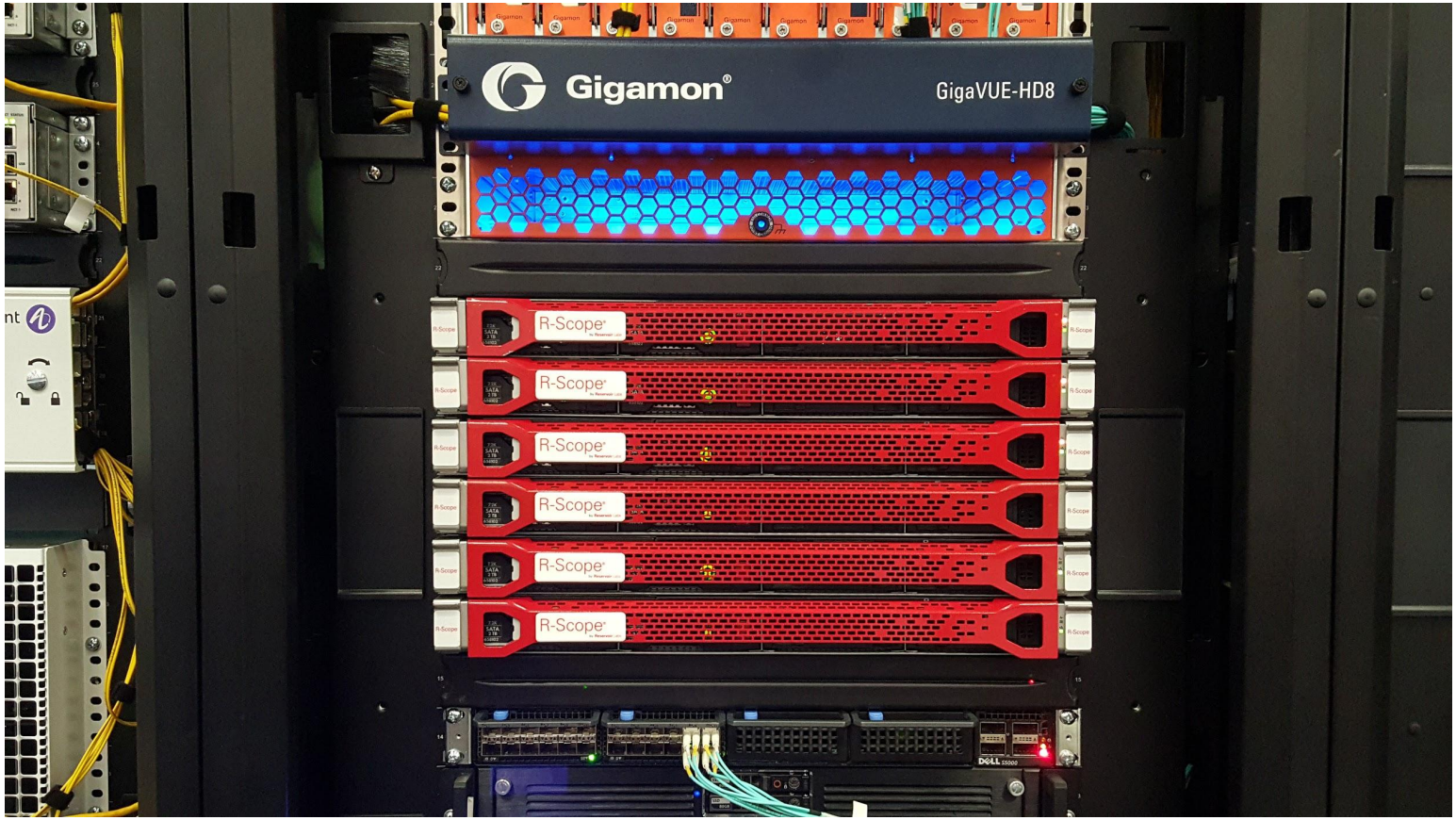
Denver to Become Epicenter for Fastest Internet and Computers in the World

Massive 1 Terabit Network to Support High Performance Computing Demonstrations at SC13 Conference



R-Scope Providing Deep Network Visibility at SC2017 / SCinet

- You can visit our demo at the SCinet NOC



Thank You

Reservoir Labs

632 Broadway
Suite 803
New York, NY 10012

812 SW Washington St.
Suite 1200
Portland, OR 97205